

```
#include "dpcagent.h" /* Our header file */

/* "fix" conflicting types */
#define AllocateResourceTag __AllocateResourceTag__
#include <advanced.h>
#undef AllocateResourceTag
#include <nwbtops.h>

#define milliclock() (GetHighResolutionTimer() / 10)

#define activityTimer ECB_DriverWorkspace.DWS_i32val

/* various flags that control the filter */
#undef FILTER_DATA_ON_RST
#define WIDEN_TCP_WINDOW
#undef TCP_ACK_LATENCY /* 10 */
/* if used at all, define *only* 1 of the following */
#undef DPCinetMaxQueuedBytes /* 4096 */
#define DPCinetMaxQueuedBytes 64
/* if defined(DPCinetMaxQueuedBytes) && defined(DPCinetMaxQueuedPackets)
#error Only 1 of DPCinetMaxQueuedBytes and DPCinetMaxQueuedPackets allowed
#endif

/* various flags that control the tunnel */
#undef TUNNEL_ONLY_TCP

#define IP_VERS(x) ((BYTE*)x)[0] >> 4)
#define IP_HD_LEN(x) ((BYTE*)x)[0] & 0x0f)
#define IP_TOS(x) ((BYTE*)x)[1]
#define IP_TOT_LEN(x) ((WORD*)x)[1]
#define IP_FLAG_FRAG(x) ((WORD*)x)[3]
#define IP_PROTO(x) ((BYTE*)x)[9]
#define IP_CSUM(x) ((WORD*)x)[5]
#define IP_SRC_ADDR(x) ((LONG*)x)[3]
#define IP_DST_ADDR(x) ((LONG*)x)[4]

#define IPPROTO_IPENCAP 0x04

#define UDP_SRC_PORT(x) ((WORD*)x)[0]
#define UDP_DST_PORT(x) ((WORD*)x)[1]

#define TCP_SRC_PORT(x) ((WORD*)x)[0]
#define TCP_DST_PORT(x) ((WORD*)x)[1]
#define TCP_ACKNUM(x) ((LONG*)x)[2]
#define TCP_CODE(x) ((BYTE*)x)[13]
#define TCP_WINDOW(x) ((WORD*)x)[7]
#define TCP_CSUM(x) ((WORD*)x)[8]

#define TCP_FIN 0x01
#define TCP_SYN 0x02
#define TCP_RST 0x04
#define TCP_PSH 0x08
#define TCP_ACK 0x10
#define TCP_URG 0x20

ECBQueue TxQ;
ECBQueue NewQ;

struct ResourceTagStructure* TxChainRTag = 0;
struct ResourceTagStructure* TxECBRRTag = 0;
LONG TxChainID;
struct ResourceTagStructure* RxChainRTag = 0;
struct ResourceTagStructure* RxECBRRTag = 0;
LONG RxChainID;
LONG DPC_IP_Address = 0;
static BYTE ConnectionMask[65536 / 8];
```

```
#ifndef __GNUC__
#define inline
#endif /* __GNUC__ */

/* ECB Manipulation */

static inline void ReleaseECB(ECB* ecb) {
    if (DIOStats) {
        if (DPCinetMaxQueuedBytes
            if ((DIOStats->QDepth -= ecb->ECB_DataLength) < 0)
                DIOStats->QDepth = 0;
        }
    }
    #ifdef DPCinetMaxQueuedPackets
        --DIOStats->QDepth;
    #endif
}

--TxECBRRTag->RTResourceCount;
CULFastSendComplete(ecb);
#ifdef LOG_ECB_ACTIVITY
    FastLogMsg(LogECBHandle, (LogClientHandle, LogECBHandle, TRUE,
        "TINET Release(%08ix)\n", ecb));
#endif /* LOG_ECB_ACTIVITY */
}

inline void Enqueue_IntsDisabled(ECBQueue* q, ECB* ecb) {
    ecb->ECB_NextLink = 0;
    if (q->tail)
        q->tail->ECB_NextLink = ecb;
    ecb->ECB_PreviousLink = q->tail;
    q->tail = ecb;
    if (q->head == 0)
        q->head = ecb;
    SignalLocalSemaphore(q->semaphore);
}

void Enqueue(ECBQueue* q, ECB* ecb) {
    _disable();
    Enqueue_IntsDisabled(q, ecb);
    _enable();
}

ECB* Dequeue(ECBQueue* q) {
    ECB* ecb;
    _disable();
    ecb = q->head;
    if (ecb == 0) {
        _enable();
        return 0;
    }
    q->head = ecb->ECB_NextLink;
    if (q->head == 0)
        q->tail = 0;
    else
        q->head->ECB_PreviousLink = 0;
    _enable();
    ecb->ECB_NextLink = ecb->ECB_PreviousLink = 0;
    return ecb;
}
```

```

void RemoveECBQueue* q, ECB* ecb) {
    _disable();
    if (ecb->ECB_NextLink)
        ecb->ECB_NextLink->ECB_PreviousLink = ecb->ECB_PreviousLink;
    else
        q->tail = ecb->ECB_PreviousLink;
    if (ecb->ECB_PreviousLink)
        ecb->ECB_PreviousLink->ECB_NextLink = ecb->ECB_NextLink;
    else
        q->head = ecb->ECB_NextLink;
    _enable();
    ecb->ECB_NextLink = ecb->ECB_PreviousLink = 0;
}

LONG InetQueuePacket(ECB* ecb, LONG board, void* chainID) {
    board = board;
    chainID = chainID;
    /* not used */
    /* not used */

    /* only handle IP packets */
    if ((*LONG*)ecb->ECB_ProtocolID != 0 ||
        *(WORD*)(ecb->ECB_ProtocolID + 4) != htons(0x0800))
        return 1;

    /* double check stats, hopefully upper layer is kosher, but */
    if (DIOSStats == 0) {
        releaseECB:
        --RxECBRTag->RTResourceCount;
        CLSULastSendComplete(ecb);
        return 0;
    }

    /* only check TCP packets to our interface */
    if (IP_PROTO(IPHeader) != IPPROTO_TCP ||
        IP_DST_ADDR(IPHeader) != DPC_IP_Address)
        return 1;

    TCPHeader = IPHeader + IP_HD_LEN(IPHeader) * 4;
    if (ecb->ECB_Fragment[0].FragmentLength < ((TCPHeader + 20) - IPHeader))
        return 1;

    if (TCP_CODE(TCPHeader) & (TCP_FIN|TCP_RST)) {
        /* release the connection */
        ClearConnection(into(TCP_DST_PORT(TCPHeader)));
    }
    else if (TCP_CODE(TCPHeader) & TCP_SYN) {
        /* allocate the connection */
        if (!AllocateConnection(into(TCP_DST_PORT(TCPHeader))))
            goto releaseECB;
    }
    return 1;
}

/* IP Manipulation */
#If 0
char *chksum (BYTE *buf, unsigned cnt)
{
    static unsigned char crc_bytes[2];
    BYTE rbl;
    WORD rax, rcx;
    int redx;
    BYTE *rdssi;

    crc_bytes[0] = crc_bytes[1] = 0;
    rcx = cnt;
    rdssi = buf;
    rbl = rcx;
    rcx = rcx >> 1;
    redx = 0;
    if (rcx != 0)
    {
        while (rcx--)
        {
            rax = *((WORD *)rdssi);
            rdssi += 2;

```

```

if (redx & 0xffff0000)
    redx++;
redx &= 0x0000ffff;
redx += rax;
}
if (redx &= 0x0000ffff)
{
    redx &= 0x0000ffff;
    redx++;
}
if (rbl & 1)
{
    rax = 0;
    rax = *rdx;
    redx += rax;
    if (redx &= 0x0000ffff)
        redx++;
}
redx = ~redx;
crc_bytes[0] = redx & 0xff;
crc_bytes[1] = (redx >> 8) & 0xff;
return (char *)crc_bytes;
};

#ifdef __GNUC__
/*
 * This is a version of ip_compute_csum() optimized for IP headers, which
 * always checksum on 4 octet boundaries.
 * This version is constructed from various places in the linux and Hughes
 * sources.
 */
static inline unsigned short ip_fold_lcomp_csum(unsigned long sum) {
    unsigned short csum;
    __asm__ ("movl %1, %w0\n\t"
            "shrl $16, %1\n\t"
            "addw %1, %w0\n\t"
            "adcl $0, %w0\n\t"
            "notw %w0"
            : "=a" (csum)
            : "b" (sum));
    return csum;
}

static inline unsigned short ip_fast_csum(unsigned short * buff, int wlen) {
    unsigned long sum = 0;
    if (wlen) {
        unsigned long eax;
        /* Suggested speedup:
        1:
        movl (%esi), %ebx
        lea (%esi+4), %esi
        adcl %ebx, %eax
        decl %ecx
        jnz 1b
        adcl $0, %eax
        movl %eax, %ebx
        shrl $16, %eax
        addw %ebx, %eax
        adcl $0, %eax
        */
    }
}

```

```

xorl $0xffff, %eax
*/
__asm__ ("clc\n\t"
        "l:\n\t"
        "lodsl\n\t"
        "adcl %3, %0\n\t"
        "loop lb\n\t"
        "adcl $0, %0\n\t"
        : "=r" (sum), "=S" (buff), "=c" (wlen), "=a" (eax)
        : "0" (sum), "1" (buff), "2" (wlen));
}
return ip_fold_lcomp_csum(sum);
}

#define chksum(b, l) ip_fast_csum(b, (l) / 4)

static inline unsigned short ip_adjust_csum(unsigned short oldcsum,
        unsigned short oldval,
        unsigned short newval) {
    unsigned long sum = ((unsigned short)~oldcsum);
    sum += ((unsigned short)~oldval);
    sum += newval;
    return ip_fold_lcomp_csum(sum);
}

#ifdef __GNUC__
/*
static int DummyFrame(FRAG_DESC* frag) { /* not used */
    frag = frag;
    return 0;
}

int (*DPCDropFrame)(FRAG_DESC* frag) = DummyFrame;

void FilterQueue(void* arg) {
    ECB* ecb;
    ECB* rover;
    BYTE* IP;
    BYTE* TCP;
    int excess;
    arg = arg; /* not used */
    RenameThread(GetThreadId(), "DPCAgent Filter");
    for (;;) {
        if (ExitingFlag)
            return;
        TimedWaitOnLocalSemaphore(NewQ.semaphore, 1000);
        if (!NewQ.head)
            continue;
        ecb = Dequeue(&NewQ);
        ecb->activityTimer = milliclock();
        IP = ecb->ECB_Fragment[0].FragmentAddress;
        if (DIOSStats == 0) {
            releaseECB:
            DPCDropFrame((FRAG_DESC*)&ecb->ECB_FragmentCount);
            --TxECBRtrTag->RTRSourceCount;
            CLSLFastSendComplete(ecb);
        }
        #ifdef LOG_ECB_ACTIVITY
            FastLogMsg(LogECBHandle, (LogClientHandle, LogECBHandle, TRUE,
                "TINET Release(%08lx)\n", ecb));
        #endif
    }
}

```

```

#endif /* LOG_ECB_ACTIVITY */
continue;
}

/* always send a fragmented or routed packet */
if (ecb->ECB_Fragment[0].FragmentLength < 20 ||
    IP_FLAG_FRAG(IP) & htons(0x3fff) ||
    IP_SRC_ADDR(IP) != DPC_IP_Address) {
    enqueueTxQ;
}
#endif
DPCInetMaxQueuedBytes
if (DIOStats->QDepth > DPCInetMaxQueuedBytes) {
    ++DIOStats->RetryTxCount;
    goto releaseECB;
}
DIOStats->QDepth += ecb->ECB_DataLength;
#endif
#endif
DPCInetMaxQueuedPackets
if (DIOStats->QDepth > DPCInetMaxQueuedPackets) {
    ++DIOStats->RetryTxCount;
    goto releaseECB;
}
++DIOStats->QDepth;
Enqueue(&TxQ, ecb);
continue;
}

excess = ecb->ECB_Fragment[0].FragmentLength - IP_HD_LEN(IP) * 4;
if (excess > 0) {
    TCP = IP + IP_HD_LEN(IP) * 4;
}
else {
    TCP = ecb->ECB_Fragment[1].FragmentAddress + (-excess);
    excess += ecb->ECB_Fragment[1].FragmentLength;
}

if (IP_PROTO(IP) != IPPROTO_UDP)
    goto filterUDP;

if (IP_PROTO(IP) != IPPROTO_TCP)
    goto enqueueTxQ;

filterTCP:
if (excess < 20)
    goto enqueueTxQ;

if (TCP_CODE(TCP) & TCP_SYN) {
    if (!AllocateConnection(nthohs(TCP_SRC_PORT(TCP))))
        goto releaseECB;
    /* scan for duplicate in TxQ */
    for (rover = TxQ.head; rover; rover = rover->ECB_NextLink) {
        BYTE* roverIP = rover->ECB_Fragment[0].FragmentAddress;
        BYTE* roverTCP;
        excess = (rover->ECB_Fragment[0].FragmentLength -
            IP_HD_LEN(roverIP) * 4);
        if (excess > 0) {
            roverTCP = roverIP + IP_HD_LEN(roverIP) * 4;
        }
        else {
            roverTCP = rover->ECB_Fragment[1].FragmentAddress + (-excess);
            excess += rover->ECB_Fragment[1].FragmentLength;
        }
        if (TCP_CODE(TCP) & (TCP_PSH | TCP_URG | TCP_FIN)) {
            ClearConnection(nthohs(TCP_SRC_PORT(TCP)));
            goto enqueueTxQ;
        }
        if (TCP_CODE(TCP) & TCP_ACK) {
            if (WIDEN_TCP_WINDOW
                WORD oldwin = TCP_WINDOW(TCP);
                WORD newwin = nthohs(oldwin);
                if (newwin < 40000) {
                    newwin += (newwin > 1);
                    newwin = nthohs(newwin);
                    TCP_WINDOW(TCP) = newwin;
                    TCP_CSUM(TCP) = ip_adjust_csum(TCP_CSUM(TCP),
                        oldwin,
                        newwin);
                }
            }
            #endif /* WIDEN_TCP_WINDOW */
            if (TCP_CODE(TCP) & (TCP_URG | TCP_PSH))
                goto enqueueTxQ;
        }
        #ifndef TCP_ACK_LATENCY
        ecb->activityTimer = milliclock() + TCP_ACK_LATENCY;
        #endif
        /* scan for redundancy in TxQ */
        for (rover = TxQ.head; rover; rover = rover->ECB_NextLink) {

```

```

TCP_CODE(roverTCP) == TCP_CODE(TCP) &&
    IP_DST_ADDR(roverIP) == IP_DST_ADDR(IP) &&
    TCP_DST_PORT(roverTCP) == TCP_DST_PORT(TCP) &&
    IP_SRC_ADDR(roverIP) == IP_SRC_ADDR(IP) &&
    TCP_SRC_PORT(roverTCP) == TCP_SRC_PORT(TCP)) {
    ++DIOStats->TxOKSingleCollision;
    goto releaseECB;
}
}
goto enqueueTxQ;
}
#define FILTER_DATA_ON_RST
if (TCP_CODE(TCP) & TCP_RST) {
    /* scan for data in TxQ */
    for (rover = TxQ; rover; rover = rover->ECB_NextLink) {
        BYTE* roverIP = rover->ECB_Fragment[0].FragmentAddress;
        BYTE* roverTCP;
        excess = (rover->ECB_Fragment[0].FragmentLength -
            IP_HD_LEN(roverIP) * 4);
        if (excess > 0) {
            roverTCP = roverIP + IP_HD_LEN(roverIP) * 4;
        }
        else {
            roverTCP = rover->ECB_Fragment[1].FragmentAddress + (-excess);
            excess += rover->ECB_Fragment[1].FragmentLength;
        }
        if (rover->ECB_Fragment[0].FragmentLength >= 20 &&
            excess >= 20 &&
            (IP_FLAG_FRAG(roverIP) & htons(0x3fff)) == 0 &&
            IP_PROTO(roverIP) == IPPROTO_TCP &&
            IP_DST_ADDR(roverIP) == IP_DST_ADDR(IP) &&
            TCP_DST_PORT(roverTCP) == TCP_DST_PORT(TCP) &&
            IP_SRC_ADDR(roverIP) == IP_SRC_ADDR(IP) &&
            TCP_SRC_PORT(roverTCP) == TCP_SRC_PORT(TCP) &&
            TCP_CODE(roverTCP) != TCP_CODE(TCP)) {
            rover->activityTimer = 0; /* will get taken out shortly */
            ++DIOStats->TxAbortCarrierSense;
        }
    }
    /* fallthru */
}
#endif
if (TCP_CODE(TCP) & (TCP_RST | TCP_FIN)) {
    ClearConnection(nthohs(TCP_SRC_PORT(TCP)));
    goto enqueueTxQ;
}
if (TCP_CODE(TCP) & TCP_ACK) {
    if (WIDEN_TCP_WINDOW
        WORD oldwin = TCP_WINDOW(TCP);
        WORD newwin = nthohs(oldwin);
        if (newwin < 40000) {
            newwin += (newwin > 1);
            newwin = nthohs(newwin);
            TCP_WINDOW(TCP) = newwin;
            TCP_CSUM(TCP) = ip_adjust_csum(TCP_CSUM(TCP),
                oldwin,
                newwin);
        }
    }
    #endif /* WIDEN_TCP_WINDOW */
    if (TCP_CODE(TCP) & (TCP_URG | TCP_PSH))
        goto enqueueTxQ;
}
#ifndef TCP_ACK_LATENCY
ecb->activityTimer = milliclock() + TCP_ACK_LATENCY;
#endif
/* scan for redundancy in TxQ */
for (rover = TxQ.head; rover; rover = rover->ECB_NextLink) {

```

```

BYTE* roverIP = rover->ECB_Fragment[0].FragmentAddress;
BYTE* roverTCP;
excess = (rover->ECB_Fragment[0].FragmentLength -
IP_HD_LEN(roverIP) * 4);
if (excess > 0) {
    roverTCP = roverIP + IP_HD_LEN(roverIP) * 4;
}
else {
    roverTCP = rover->ECB_Fragment[1].FragmentAddress + (-excess);
    excess += rover->ECB_Fragment[1].FragmentLength;
}
if (rover->ECB_Fragment[0].FragmentLength >= 20 &&
    excess >= 20 &&
    (IP_FLAG_FRAG(roverIP) & htons(0x3fff)) == 0 &&
    IP_PROTO(roverIP) == IPPROTO_TCP &&
    IP_DST_ADDR(roverIP) == IP_DST_ADDR(IP) &&
    TCP_DST_PORT(roverTCP) == TCP_DST_PORT(TCP) &&
    IP_SRC_ADDR(roverIP) == IP_SRC_ADDR(IP) &&
    TCP_SRC_PORT(roverTCP) == TCP_SRC_PORT(TCP) &&
    TCP_CODE(roverTCP) & TCP_ACK &&
    htonl(TCP_ACKNUM(roverTCP)) + htons(TCP_WINDOW(roverTCP)) <
    htonl(TCP_ACKNUM(TCP)) + htons(TCP_WINDOW(TCP))) {
    /* move ACK information over to TxQ and release this packet */
    TCP_CSUM(roverTCP) = ip_adjust_csum(TCP_CSUM(roverTCP),
    TCP_WINDOW(roverTCP),
    TCP_WINDOW(TCP));
    TCP_CSUM(roverTCP) = ip_adjust_csum(TCP_CSUM(roverTCP),
    (WORD)TCP_ACKNUM(roverTCP),
    (WORD)TCP_ACKNUM(TCP));
    TCP_CSUM(roverTCP) = ip_adjust_csum(TCP_CSUM(roverTCP),
    TCP_ACKNUM(roverTCP)>>16,
    TCP_ACKNUM(TCP)>>16);
    TCP_ACKNUM(roverTCP) = TCP_ACKNUM(TCP);
    TCP_WINDOW(roverTCP) = TCP_WINDOW(TCP);
    ++DIOSStats->TxAbortExcessCollisions;
    goto releaseECB;
}
goto enqueueTxQ;
}
goto enqueueTxQ;

filterUDP:
{
    BYTE* UDP = TCP;
    BYTE* DNS;

    /* ECB contents determined by inspection, there are safer methods */
    if (excess < 8)
        goto enqueueTxQ;

    /* filter DNS only */
    if (UDP_DST_PORT(UDP) != htons(53))
        goto enqueueTxQ;

    excess -= 8;
    DNS = (excess > 0) ? (UDP + 8) : ecb->ECB_Fragment[1].FragmentAddress;
    for (rover = TxQ.head; rover; rover = rover->ECB_NextLink) {
        BYTE* roverIP = rover->ECB_Fragment[0].FragmentAddress;
        BYTE* roverUDP;
        BYTE* roverDNS;
        excess = (rover->ECB_Fragment[0].FragmentLength -
            IP_HD_LEN(roverIP) * 4);
        if (excess > 0) {
            roverUDP = roverIP + IP_HD_LEN(roverIP) * 4;

```

```

}
else {
    roverUDP = rover->ECB_Fragment[1].FragmentAddress + (-excess);
    excess += rover->ECB_Fragment[1].FragmentLength;
}
if (rover->ECB_Fragment[0].FragmentLength >= 20 &&
    excess >= 8 &&
    (IP_FLAG_FRAG(roverIP) & htons(0x3fff)) == 0 &&
    IP_PROTO(roverIP) == IPPROTO_UDP &&
    IP_DST_ADDR(roverIP) == IP_DST_ADDR(IP) &&
    UDP_DST_PORT(roverUDP) == UDP_DST_PORT(UDP) &&
    IP_SRC_ADDR(roverIP) == IP_SRC_ADDR(IP) &&
    UDP_SRC_PORT(roverUDP) == UDP_SRC_PORT(UDP) &&
    (roverDNS = (((excess -= 8) > 0) ?
        (roverUDP + 8) :
        rover->ECB_Fragment[1].FragmentAddress)) &&
    * (LONG*)DNS == * (LONG*)roverDNS) {
    ++DIOSStats->TxAbortLateCollision;
    goto releaseECB;
}
goto enqueueTxQ;
}
}

/* SLIP, PPP, Modem Manipulation */
#define MAX_READ_BUF 128

int InetState = MODEM_IDLE;
static BYTE SlipEndPkt[1] = (END);

int WaitingLines = 0, NextWait = 0;
char WaitingBuffer[MAX_READ_BUF];
int WaitingIndex = 0;
LONG ConnectingTimeout = 0;
LONG ConnectingRedial = FALSE;

int BaudRate[] =
{
    2400, /* 0 */
    3600, /* 1 */
    4800, /* 2 */
    7200, /* 3 */
    9600, /* 4 */
    19200, /* 5 */
    38400, /* 6 */
    57600, /* 7 */
    115200 /* 8 */
};

void InitLogin()
{
    int i;
    char *nextWait;

    WaitingLines = 0;
    if (DlCfg.auto_login)
    {
        WaitingIndex = 0;
        WaitingBuffer[WaitingIndex] = '\0';
        NextWait = 0;
    }
}

```

```

ConnectingTimeout = 0;
for (i = 0, nextWait = DloCfg.wait_for_1; i < 9; i++, nextWait =
sizeof(DloCfg.wait_for_1))

```

```

    if (*nextWait)
        WaitingLines++;
}

```

```

static BYTE MTUBuffer[8192];

```

```

int SLIPSendRoutineOpt(FRAG_DESC* fragStruc)

```

```

{
    LONG count = 0;
    BYTE* output = MTUBuffer;

    *output++ = END;
    while (count < fragStruc->FragmentCount)
    {
        FRAGMENTSTRUCT* frag = fragStruc->FragmentDesc + count;
        BYTE* frame = (BYTE*)frag->FragmentAddress;
        LONG length = frag->FragmentLength;

        while (length-- > 0)
        {
            switch (*frame)
            {
                case END:
                    *output++ = ESC;
                    *output++ = ESC_END;
                    break;
                case ESC:
                    *output++ = ESC;
                    *output++ = ESC_ESC;
                    break;
                default:
                    *output++ = *frame;
                    break;
            }
            ++count;
        }
    }
}

```

```

    while (length-- > 0)
    {
        switch (*frame)
        {
            case END:
                *output++ = ESC;
                *output++ = ESC_END;
                break;
            case ESC:
                *output++ = ESC;
                *output++ = ESC_ESC;
                break;
            default:
                *output++ = *frame;
                break;
        }
        ++frame;
    }
    ++count;
}
*output++ = END;

```

```

if (output - MTUBuffer < 22 ||
    output - MTUBuffer > DloGetWriteBufferSize())
    return 0;
DloSend(MTUBuffer, output - MTUBuffer, DLO_INET_TIMEOUT);
return 1;
}

```

```

int SLIPSendRoutineDebug(FRAG_DESC* fragStruc)

```

```

{
    LONG count = 0;
    BYTE* output = MTUBuffer;
    BYTE* dataStart = 0; /* separate HEADER from PAYLOAD */
    LONG header = 0x80000000;

    while (count < fragStruc->FragmentCount)
    {

```

```

FRAGMENTSTRUCT* frag = fragStruc->FragmentDesc + count;
BYTE* frame = (BYTE*)frag->FragmentAddress;
LONG length = frag->FragmentLength;

```

```

while (length-- > 0)
{
    switch (*frame)
    {

```

```

        case END:
            *output++ = ESC;
            *output++ = ESC_END;
            break;
        case ESC:
            *output++ = ESC;
            *output++ = ESC_ESC;
            break;
        default:
            *output++ = *frame;
            break;
    }
    if (header == 0x80000000)
        if (header = (*frame & 0x0f) * 4;
        if (--header == 0)
            dataStart = output;
            ++frame;
        }
        ++count;
    }
}

```

```

if (output - MTUBuffer < 20 ||
    output - MTUBuffer > DloGetWriteBufferSize())
    return 0;
DloSend(SlipEndpkt, 1, DLO_INET_TIMEOUT);
DloSend(MTUBuffer, dataStart - MTUBuffer, DLO_INET_TIMEOUT);
DloSend(dataStart, output - dataStart, DLO_INET_TIMEOUT);
DloSend(SlipEndpkt, 1, DLO_INET_TIMEOUT);
return 1;
}

```

```

int (*DPCtxFrame)(FRAG_DESC* fragStruc) = SLIPSendRoutineOpt;

```

```

/*****
*
* IPSendRoutine(ECB *tcb)
*
* Description:
*
* Input:      ecb
*
* Control Block
*
* Output:     nothing
*
* Returns:    0 if finished with ECB
*
*****/

```

```

*
* IPSendRoutine(ECB *tcb)
*
* Description:
*
* Input:      ecb
*
* Control Block
*
* Output:     nothing
*
* Returns:    0 if finished with ECB
*
*****/

```

```

*
* IPSendRoutine(ECB *tcb)
*
* Description:
*
* Input:      ecb
*
* Control Block
*
* Output:     nothing
*
* Returns:    0 if finished with ECB
*
*****/

```

```

*
* IPSendRoutine(ECB *tcb)
*
* Description:
*
* Input:      ecb
*
* Control Block
*
* Output:     nothing
*
* Returns:    0 if finished with ECB
*
*****/

```

```

*
* IPSendRoutine(ECB *tcb)
*
* Description:
*
* Input:      ecb
*
* Control Block
*
* Output:     nothing
*
* Returns:    0 if finished with ECB
*
*****/

```

```

*
* IPSendRoutine(ECB *tcb)
*
* Description:
*
* Input:      ecb
*
* Control Block
*
* Output:     nothing
*
* Returns:    0 if finished with ECB
*
*****/

```

```

*
* IPSendRoutine(ECB *tcb)
*
* Description:
*
* Input:      ecb
*
* Control Block
*
* Output:     nothing
*
* Returns:    0 if finished with ECB
*
*****/

```

```

*
* IPSendRoutine(ECB *tcb)
*
* Description:
*
* Input:      ecb
*
* Control Block
*
* Output:     nothing
*
* Returns:    0 if finished with ECB
*
*****/

```

```

*
* IPSendRoutine(ECB *tcb)
*
* Description:
*
* Input:      ecb
*
* Control Block
*
* Output:     nothing
*
* Returns:    0 if finished with ECB
*
*****/

```

```

static BYTE IPHeader[IP_TUNNEL_SIZE] =
{
    0x45, /* version 4, length 5 */
    0, /* tos */
    0, 0, /* length */
}

```

```

0, 0, /* ident */
0, 0, /* fragment */
0x7f, /* ttl */
4, /* IP in IP (encapsulation) */
);
#define IPHeaderIdent (*(WORD*)&IPHeader[4])
int
IPSendRoutine(ECB *ecb)
{
    FRAG_DESC* fragStruc = alloca(sizeof(LONG) + (sizeof(FRAGMENTSTRUCT) *
    (ecb->ECB_FragmentCount + 2)));
    WORD frame_size = ecb->ECB_DataLength;
    int options_collapsed = 1;
    LONG currFrag = 0;
    BYTE* ecbIPHeader = ecb->ECB_Fragment[0].FragmentAddress;

    /* initialize the copy of the tcb fragStruc */
    memcpy(fragStruc,
    &ecb->ECB_FragmentCount,
    sizeof(LONG) + (sizeof(FRAGMENTSTRUCT) * ecb->ECB_FragmentCount));

    if (frame_size < DloCfg.mtu &&
    TUNNEL_ONLY_TCP
    /* UDP doesn't need tunnel header */
    (ecbIPHeader[9] != IPPROTO_TCP) ||
    /* either do "routed" packets */
    ((*LONG*)&ecbIPHeader[12] != DPC_IP_Address)))
        goto skipFragger;

    memset(&fragStruc->FragmentDesc(fragStruc->FragmentCount),
    0,
    sizeof(FRAGMENTSTRUCT) * 2);

    frame_size += IP_TUNNEL_SIZE;

    /* fill IPHeader with tunnel data, including IP/gateway addresses,
    * and prepend to frag list.
    */
    *(WORD*)&IPHeader[2] = htons(frame_size);
    ++IPHeaderIdent;
    *(WORD*)&IPHeader[10] = 0; /* checksum, for now */
    *(LONG*)&IPHeader[12] = DloCfg.ip_address;
    *(LONG*)&IPHeader[16] = DloCfg.gateway_address;
    memmove(fragStruc->FragmentDesc + 1,
    fragStruc->FragmentDesc,
    sizeof(FRAGMENTSTRUCT) * fragStruc->FragmentCount);
    fragStruc->FragmentDesc[0].FragmentAddress = IPHeader;
    fragStruc->FragmentDesc[0].FragmentLength = IP_TUNNEL_SIZE;
    ++fragStruc->FragmentCount;
    ++currFrag;
    *(WORD*)&IPHeader[10] = chksum((WORD *)IPHeader,
    IP_TUNNEL_SIZE);

    while (frame_size > DloCfg.mtu)
    {
        /*
        * Shucks. Have to fragment the packet.
        * This algorithm is roughly per RFC791.
        */
        LONG OIHL = fragStruc->FragmentDesc[0].FragmentLength;
        BYTE OMF = IPHeader[6] & 0x20;
        LONG NPH (DloCfg.mtu - OIHL) & 0xffff;
        WORD TL = OIHL + NPH;

        IPHeader[6] |= 0x40 | (fragStruc->FragmentDesc[0].Fragment
        ntLength;
        ntLength;

        options_collapsed = 1;
    }
}

IPHeader[6] |= 0x20; /* set More Fragments */
*(WORD*)&IPHeader[2] = htons(TL);
*(WORD*)&IPHeader[10] = 0; /* clear checksum */
*(WORD*)&IPHeader[10] = chksum((WORD *)IPHeader, OIHL);

/*
* Now fake out the fragStruc to reflect TL.
* Hang on to enough information to remove the TL less OIHL
* later.
*/
TL -= OIHL;
frame_size -= TL;
while (TL > 0 &&
    fragStruc->FragmentDesc(currFrag).FragmentLength <= TL)
{
    TL -= fragStruc->FragmentDesc(currFrag).FragmentLength;
    ++currFrag;
}
if (TL > 0)
{
    /*
    * This frag gets split into 2 pieces.
    */
    memmove(fragStruc->FragmentDesc + currFrag + 1,
    fragStruc->FragmentDesc + currFrag,
    sizeof(FRAGMENTSTRUCT) *
    (fragStruc->FragmentCount - currFrag));
    ++fragStruc->FragmentCount;
    fragStruc->FragmentDesc[currFrag].FragmentLength = TL;
    ++currFrag;
    fragStruc->FragmentDesc[currFrag].FragmentLength -= TL;
    fragStruc->FragmentDesc[currFrag].FragmentAddress = ((ch
    ar*)fragStruc->FragmentDesc(currFrag).FragmentAddress) + TL;
}
TL = fragStruc->FragmentCount - currFrag + 1;
fragStruc->FragmentCount = currFrag;

if (DPCtxFrame(fragStruc) == 0)
    return 0;

if (!options_collapsed) {
    LONG offset = 20;
    while (offset < OIHL && IPHeader[offset]) {
        if (IPHeader[offset] & 0x80) /* copy */
            offset += IPHeader[offset + 1];
        else /* collapse */
            LONG len = IPHeader[offset + 1];
            memcpy(IPHeader + offset,
            IPHeader + offset + len,
            OIHL - (offset + len));
            OIHL -= len;
        }
    }
    offset = fragStruc->FragmentDesc[0].FragmentLength;
    fragStruc->FragmentDesc[0].FragmentLength = (OIHL + 3) &
    0x3c;
    memset(IPHeader + OIHL,
    0,
    fragStruc->FragmentDesc[0].FragmentLength - OIHL);
    IPHeader[0] = 0x40 | (fragStruc->FragmentDesc[0].Fragment
    ntLength / 4);
    frame_size -= offset - fragStruc->FragmentDesc[0].Fragme
    ntLength;
    options_collapsed = 1;
}

```

```

/* Adjust the frag list to "remove" the fragment just sent.
 */
memmove(fragStruc->FragmentDesc + 1,
        fragStruc->FragmentDesc + currFrag,
        sizeof(FRAGMENTSTRUCT) * TL);
fragStruc->FragmentCount = TL;
currFrag = 1;

/* compute new IPHeader values: fragment offset */
*(WORD*)(&IPHeader[6]) = htons((ntohs(*(WORD*)(&IPHeader[6])) &
0x1fff)
+ (NFB / 8));

IPHeader[6] |= OMF;
if (frame_size <= DloCfg.mtu)
{
    *(WORD*)(&IPHeader[2]) = htons(frame_size);
    *(WORD*)(&IPHeader[10]) = 0; /* clear checksum */
    *(WORD*)(&IPHeader[10]) = chksum((WORD*)IPHeader,
        fragStruc->FragmentDesc
        [0].FragmentLength);
    break;
}

skipFragger:
/* send the remaining (possibly ALL) portion of the frame */
if (DPCTXFrame(fragStruc) == 0)
    return 0;

if (DIOStats)
{
    ++DIOStats->TotalTxPacketCount;
    if ((DIOStats->TotalTxOKByteCountLow += ecb->ECB_DataLength) <
        ecb->ECB_DataLength)
        ++DIOStats->TotalTxOKByteCountHigh; /* wrapped */
}
return 1;

static void EmptyESR(ECB* ecb) {
}

unsigned char RawEnvelope[14] = {
    0x00, 0x00, 0x0c, 0x0a, 0x0b, 0x0c,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x08, 0x00,
};

ECB RawECB = {
    0, /* ECB_NextLink */
    0, /* ECB_PreviousLink */
    0, /* ECB_Status */
    EmptyESR, /* ECB_ESR */
    -1, /* ECB_StackID */
    "", /* ECB_ProtocolID */
    0, /* ECB_BoardNumber */
    "", /* ECB_ImmediateAddress */
    { 0 }, /* ECB_DriverWorkspace */
    { 0 }, /* ECB_ProtocolWorkspace */
    -1, /* ECB_DataLength */
    -1, /* ECB_FragmentCount */
    ( RawEnvelope, sizeof(RawEnvelope) ),
};

```

```

};

#define RawSendRoutineDebug RawSendRoutineOpt
int RawSendRoutineOpt(FRAG_DESC* fragStruc) {
    int i;
    for (i = 20000; --i > 0; ) {
        if (RawECB.ECB_Status == 0)
            goto rawSend;
        if (ExitingFlag)
            return 0;
        ThreadSwitchWithDelay();
    }
    return 0;
rawSend:
    i = fragStruc->FragmentCount;
    RawECB.ECB_FragmentCount = i + 1;
    memcpy(RawFrag,
        fragStruc->FragmentDesc,
        i * sizeof(FRAGMENTSTRUCT));
    RawECB.ECB_DataLength = sizeof(RawEnvelope);
    while (i > 0)
        RawECB.ECB_DataLength += RawFrag[--i].FragmentLength;
    CLSLSendPacket(&RawECB);
    return 1;
}

void DisplayWaitStatus(void)
{
    char statusStr[80];
    char *waitingStr;

    if (WaitingLines == 0)
        return;

    waitingStr = (char *)&DloCfg.wait_for_1[NextWait*30];
    NWSprintf(statusStr, MSG("Modem Status: Waiting for \"%s\"\\n", 557), wait
        ingStr);
    UpdateModemStr(statusStr);
}

void InetStateChange(int state) {
    if (DloCfg.out_protocol == OUT_NETWORK) {
        InetState = PROTOCOL_CONNECTED;
        return;
    }
    switch (state) {
        case DLOS_IDLE:
        case DLOS_DISC_1:
        case DLOS_DISC_2:
        case DLOS_DISC_3:
        case DLOS_DISC_4:
            InetState = MODEM_IDLE;
            if (ConnectingRedial) {
                DloEndConn();
                ConnectingRedial = FALSE;
            }
            break;
        case DLOS_CONN:
            InetState = MODEM_CONNECTED;
            InitLogin();
            if (InetAsleep)
                ResumeThread(DPCInetPID);
            break;
    }
}

```

```

/* technically unsafe, but works */
IPHeader, sizeof(IPHeader),
};

```



```

default:
break;
)
)

/*****
* FUNCTION: Convert Internet address Address
* DESCRIPTION: converts a character string containing the Internet address
* into a form that BIC DP understands.
* e.g. 139.85.124.06 (8B.55.7C.06) into 067C558B0000
*****/
void convert_address(char *lpszIpAddress)
{
    char *p;
    int i = 0;
    char tmp[20], tmp1[10];
    tmp[0] = 0;
    while((p=strchr(lpszIpAddress, (int)('.'))) != NULL)
    {
        i = atoi(p+1);
        NWSprintf(tmp1, MSG("%02X", 477), i);
        strcat(tmp, tmp1);
        *p = 0;
    }
    i = atoi(lpszIpAddress);
    NWSprintf(tmp1, MSG("%02X", 478), i);
    strcat(tmp, tmp1);
    strcat(tmp, MSG("0000", 479));
    CStrCpy(lpszIpAddress, tmp);
}

MACAddr_t      HIAddr;
LONG            InetChannel;

void make_hi_key(chunk *key)
{
    int i;
    LONG sn;
    BYTE serialNum[9];
    BYTE serialNumPacked[3];
    BYTE x;

    DIOGetSN(serialNum);
    sn = atoi(serialNum);
    NWSprintf(serialNum, MSG("%06lx", 480), sn);

    pack_mac_addr(serialNumPacked, 3, serialNum, 6);
    x = serialNumPacked[0];
    serialNumPacked[0] = serialNumPacked[2];
    serialNumPacked[2] = x;

    key->b[0] = serialNumPacked[0] ^ 0xff;
    key->b[1] = serialNumPacked[1] ^ 0xff;
    key->b[2] = serialNumPacked[2] ^ 0xff;
    for(i = 3; i < 8; i++)
        key->b[i] = 0x00 ^ 0xff;

    MACbuildAddr(serialNum, MAC_HI, 0, &HIAddr);
}

void InetChangeProtocol(void)
{
    switch (DioCfg.out_protocol) {
    case OUT_PPP:
        DPCTXFrame = DebugFlag ? PPPSendRoutineDebug : PPPSendRoutineOpt;
        break;
    case OUT_NETWORK:
        void (*ControlEntryPoint)(void) = 0;
        struct DriverConfigurationStructure* dvrCfg = 0;
        if (CISLGetMLIDControlEntry(DioCfg.net_interface,
                                   &ControlEntryPoint))
        {
            goto skipDriver;
        }
        dvrCfg = (struct DriverConfigurationStructure *)
            CommandMlid(DioCfg.net_interface, 0, (LONG)ControlEntryP
            memcpy(RawEnvelope + 6, dvrCfg->DNodeAddress, 6);

        skipDriver:
        RawECB_ECB_BoardNumber = DioCfg.net_interface;
        memcpy(RawEnvelope, DioCfg.net_addr, 6);
        DPCTXFrame = DebugFlag ? RawSendRoutineDebug : RawSendRoutineOpt;
        break;
    case OUT_SLIP:
        DPCTXFrame = DebugFlag ? SLIPSendRoutineDebug : SLIPSendRoutineOpt;
        break;
    }
    InetStateChange(DLOS_DISC_4);
    DioEndConn();
}

int ProcessLogin(void)
{
    BYTE value;
    char *sendStr, *waitStr;
    char sendBuf[40];
    LONG nextTimeout;

    /* No use trying if we aren't even connected */
    /* Get out if we're done */
    if (WaitingLines == 0 || DioCfg.auto_login == FALSE)
    {
        return(TRUE);
    }
    if (!DioConnected())
        return(FALSE);
    /* Timeout if we've waited too long for this wait */
    if (ConnectingTimeout == 0)
    {
        ConnectingTimeout = GetCurentTime() + DioCfg.wait_timeout * 1
    }
    if (GetCurentTime() > ConnectingTimeout)

```

```

    if (ConnectingRedial == FALSE)
    {
        /* First timeout. Send return and try again. */
        ConnectingRedial = TRUE;
        InitLogin();
        DloSend(MSG("\r", 181), 1, DLO_INET_TIMEOUT);
        return(FALSE);
    }
    DloEndConn();
    return(FALSE);
}

DisplayWaitStatus();

while (DloReceive(&value, 1) != 0)
{
    if (DebugFlag)
        putchar(value);
    if (value != '\r' && value != '\n')
    {
        WaitingBuffer[WaitingIndex++] = value;
        WaitingBuffer[WaitingIndex] = 0;
        if (WaitingIndex > (MAX_READ_BUF-1))
            WaitingIndex = 0;
    }

    waitStr = (char *)DloCfg.wait_for_1(NextWait * 30);
    if (strstr(WaitingBuffer, waitStr) != NULL)
    {
        sendStr = (char *)DloCfg.send_1(NextWait * 30);
        NWSprintf(sendBuf, MSG("%s\r", 558), sendStr);
        DloSend(sendBuf, CStrLen(sendBuf), DLO_INET_TIMEOUT);
        NextWait++;
        WaitingIndex = 0;
        WaitingBuffer[WaitingIndex] = '\0';
        WaitingLines--;
        if (WaitingLines == 0)
        {
            DloUpdateModemStr();
            return(TRUE);
        }
        DisplayWaitStatus();
        nextTimeout = DloCfg.wait_timeout_1 + NextWait;
        ConnectingTimeout = GetCurrentTime() +
            ((nextTimeout) ? (nextTimeout * 18) : (5
*18));
    }
    else if (value == '\r')
    {
        WaitingIndex = 0;
        WaitingBuffer[WaitingIndex] = '\0';
    }
}

return(FALSE);

int ConnectProtocol(void)
{
    int ccode;

    if (DloCfg.out_protocol == OUT_SLIP)
    {
        delay(1000);
        return 1;
    }
}

```

```

void TinetProtocolBind(LONG __parameter) {
    struct EventProtocolBindStruct* epbs =
    (struct EventProtocolBindStruct*)__parameter;
    if (epbs->boardNumber == DIOBoard &&
        epbs->protocolNumber == 1/*PROTOCOL_ID_TCPPIP*/) {
        extern LONG DPCNextRegistrationCheck;
        DPCGetIPAddress(&DPC_IP_Address);
        DPCNextRegistrationCheck = 0;
    }
}

/******
 *
 * InetMain(void *parm)
 *
 * Description:
 *     Main thread for Turbo Internet handling.
 *
 * Input:
 *     parm
 *
 * Output:
 *     nothing
 *
 * Returns:
 *     nothing
 *
 ******
 */

void InetMain(void *parm)
{
    time_t nextStartConn = 0;
    LONG removedCount = (LONG)(-1);
    long millisecondDelay = 0;
    LONG protocolBindHandle =
        RegisterForEvent(EVENT_PROTOCOL_BIND, TinetProtocolBind, 0);

    parm = parm; /* unused */

    NewQ.semaphore = OpenLocalSemaphore(0);
    TxQ.semaphore = OpenLocalSemaphore(0);
    BeginThread(FilterQueue, 0, 0, 0);

    TxChainRTag = AllocateResourceTag(NLMHandle,
        MSG("Turbo Inet TxPrescan Chain", 476)
        LSLTxPrescanStackSignature);
    TxECBRTag = AllocateResourceTag(NLMHandle,
        MSG("Turbo Inet Transmit Packets", 619),
        ECBSignature);
    RxChainRTag = AllocateResourceTag(NLMHandle,
        "Turbo Inet RxPrescan Chain",
        LSLRxPrescanStackSignature);
    RxECBRTag = AllocateResourceTag(NLMHandle,
        MSG("Turbo Inet Receive Packets", 619),

```

```

ECBSignature);
DPCGetIPAddress(&DPC_IP_Address);
if (DlCfg.out_protocol == OUT_NETWORK)
    InetState = PROTOCOL_CONNECTED;

mainloop: while (!ExitingFlag)
{
    InetAsleep = TRUE;
    if (millidelay > 55)
        delay(millidelay);
    else if (millidelay > 0)
        ThreadSwitchWithDelay/*LowPriority*/();
    else
        ThreadSwitch();
    InetAsleep = FALSE;
    while (DIOBoard && removedCount != DIORemovedCount)
    {
        BYTE address[8];
        BYTE szBicBCDAddress[20];
        struct DriverStatsStructure* stats = 0;
        LONG ip_address = ntohs(DlCfg.ip_address);
        removedCount = DIORemovedCount;
        /* Enable internet reception */

        /* Yuk. We'll change this later to get rid these extra s
        NWSprintf(szBicBCDAddress, MSG("%d.%d.%d.%d", 620),
            (ip_address >> 24) & 0xff,
            (ip_address >> 16) & 0xff,
            (ip_address >> 8) & 0xff,
            (ip_address) & 0xff);
        convert_address(szBicBCDAddress);
        if (!pack_mac_addr(address, 6,
            szBicBCDAddress, CStrLen(szBicBCDAddr
ess)))
        {
            /* UpdateModemStr(MSG("ERROR: could not pack mac
            address
            \n", xxxx)); */
            millidelay = 500;
            break;
        }

        /* Sending an esr address of -1 tells MUID to handle rec
        if (DIOOpenChannel(address,
            (int (*)())0xffffffff,
            &InetChannel))
        {
            millidelay = 500;
            removedCount = (LONG)(-1);
            break;
        }
        if (ExitingFlag)
            break;
        DIOAddHIAddr(InetChannel, (BYTE *)&HIAddr);
        DPCGetMUIDStats(&stats);
        DIOWriteStatus(AIOPortHandle, &count, 0);
        if (CLSLRegisterPreScanTxChain(TxChainRtag,
            DIOBoard,
            3, /* next to last */
            &TxChainID,
            InetQueuePacket,
            InetControl,
            TxECBRtag))
        {
            millidelay = 500;
            removedCount = (LONG)(-1);
            break;
        }
        if (CLSLRegisterPreScanRxChain(RxChainRtag,
            DIOBoard,
            3, /* next to last */
            &RxChainID,
            ConnectionLimiter,
            InetControl,
            RxECBRtag))
        {
            millidelay = 500;
            removedCount = (LONG)(-1);
            break;
        }
        if (DlCfg.out_protocol == OUT_PPP &&
            InetState == PROTOCOL_CONNECTED)
        {
            PPPBackground();
        }
        if (TxQ.head == 0 &&
            (InetState <= MODEM_CONNECTING ||
            InetState >= PROTOCOL_CONNECTED))
        {
            TimedWaitOnLocalSemaphore(TxQ.semaphore, 200);
            millidelay = 0;
            continue;
        }
        switch (InetState)
        {
            case MODEM_CONNECTED:
                if (!ProcessLogin())
                {
                    millidelay = 500;
                    break;
                }
                InetState = LOGIN_CONNECTED;
                /* fallthru */
            case LOGIN_CONNECTED:
                if (!ConnectProtocol())
                {
                    DIOEndConn();
                    millidelay = 15 * 1000;
                    break;
                }
                InetState = PROTOCOL_CONNECTED;
                millidelay = 1;
                break;
            case PROTOCOL_CONNECTED:
                LONG count = 0;
                if (DlCfg.out_protocol != OUT_NETWORK &&
                    AIOWriteStatus(AIOPortHandle, &count, 0))
                {
                    millidelay = 200;
                    break;
                }
        }
    }
}

```

```

    if (count == 0)
    {
        LONG milliclock = milliclock();
        ECB* ecb;
        ecb = TXQ.head;
        millidelay = 100;
        while (ecb->activityTimer > milliclock)
        {
            LONG diff = ecb->activityTimer - milliclock;

            if (diff < millidelay)
                millidelay = diff;
            if ((ecb = ecb->ECB_NextLink) == 0)
                goto mainloop;
        }
        Remove(&TXQ, ecb);
        if (ecb->activityTimer < milliclock() - 60000) {
            if (ecb->activityTimer)
                ++DIOWriteStatus(AIOPortHandle, &count, 0);
            else
                IPSendRoutine(ecb);
            ReleaseECB(ecb);
            if (DIOcfg.out_protocol != OUT_NETWORK)
                AIOWriteStatus(AIOPortHandle, &count, 0);
        }
        millidelay = (count *
            10 * /* Tx bits with framing */
            1000 / /* milliseconds */
            BaudRate(DIOcfg.tinet_baud_index));
        break;
    }

    case MODEM_IDLE:
        if (nextStartConn < time(0))
        {
            InitLogin();
            DIOStartConn(DIO_INET_TIMEOUT);
            InetState = MODEM_CONNECTING;
            nextStartConn = time(0) + 30;
            /* fallthru */
        }

        default:
            millidelay = 10 * 1000;
            break;
        }

    DIOCloseChannel(InetChannel);

    CLSLDeRegisterPreScanRxChain(RxChainID);
    CLSLDeRegisterPreScanTxChain(TxChainID);
    while (TXQ.head)
        ReleaseECB(Dequeue(&TXQ));
    while (NewQ.head)
        ReleaseECB(Dequeue(&NewQ));
    CloseLocalSemaphore(TxQ.semaphore); TxQ.semaphore = 0;
    CloseLocalSemaphore(NewQ.semaphore); NewQ.semaphore = 0;

    UnregisterForEvent(protocolBindHandle);

    DPCTinetPID = 0;
    return;
}

```